UTEC-CSc-79-039

Copy # 36

Semi-Annual Technical Report

LEVEL II

NOISE SUPPRESSION METHODS FOR ROBUST SPEECH PROCESSING

Contractor:	University of Utah
Effective Date:	2 January 1979
Expiration Date:	30 September 1980
Reporting Period:	1 October 1978 – 31 March 1979

Principal Investigator: Dr. Steven F. Boll

Telephone: (801) 581-8224

Sponsored by

Defense Advanced Research Projects Agency (DoD)

ARPA Order No. 3301

Monitored by Naval Research Laboratory

Under Contract No. N00173-79-C-0045

April 1979

12

1 Dec 78 - 28 Feb 79

(9) Quarterly Research and Development Technical Report (6) Spatial Data Management System

Contract Period Covered by Report: 1 December 1978 – 28 Feb 1979 (11)

(12) 85 p.

Computer Corporation of America, Cambridge, Ma.

(10) Christopher F. Herot
David Kramlich
Richard Carling
Mark Friedell
Jerry Farrell

Research Division
Computer Corporation of America

The views and conclusions in this document are those of the authors and should not be interpreted as necessarily representing the official policies, express or implied, of the Advanced Research Projects Agency, or the United States Government.

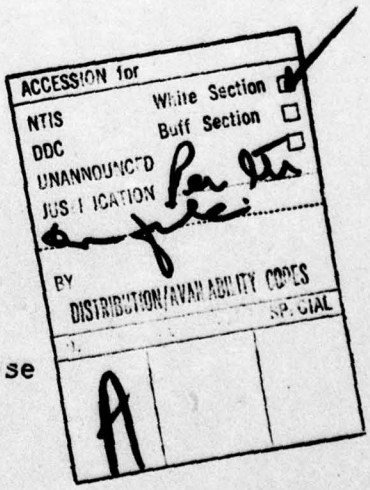
Sponsor: Defense Advanced Research Projects Agency
Office of Cybernetics Technology

ARPA Order Number: 3487

(15) ARPA Contract Number: MDA903-78-C-0122

Contract Period: 15 February 1978 – 30 November 1979

Table of Contents



1. INTRODUCTION

This fifth quarter of work on the design and implementation of a prototype Spatial Data Management System (SDMS) resulted in the addition of several new capabilities ~~which~~ to the operational prototype. These capabilities provide:

(1) a means of controlling the detail at which data is presented; and

(2) additional flexibility in the appearance of graphical displays of symbolic data.

In addition, a mechanism for maintaining the correspondence between symbolic and graphical forms of data was designed and will be implemented during the coming two quarters.

1.1 Graphical Data Spaces

The Spatial Data Management System (SDMS) provides the user with a large electronic work space referred to as the Graphical Data Space (GDS). The GDS is composed of one or more flat surfaces called Information Spaces, or I-Spaces upon which information can be displayed. This information can originate in either graphical or symbolic form.

Graphical data is entered via an interactive "painting" program which allows the user to define shapes, colors, and text. Symbolic data is fetched from a relational database management system and used to generate icons which are graphical representations of entities in the symbolic database.

The user of an SDMS can view the data through a set of color displays as shown in Figure 1. The leftmost of the three monitors shows the current I-Space in its entirety. This display serves as a world view map for helping the user find his way around the data. In the photograph, the user is examining a database of ships which originated as symbolic data in a relational database management system.

The main display in the center of the configuration shows a magnified view of the I-Space. The portion of the I-Space currently being magnified is shown as a highlighted rectangle on the world view map. By pressing on a joy stick (visible in the user's left hand in Figure 1), this highlighted rectangle can be made to move across the I-Space. To a user watching the main display, the effect is that of moving in a plane parallel to the data surface. This motion can be used to see various parts of the database.

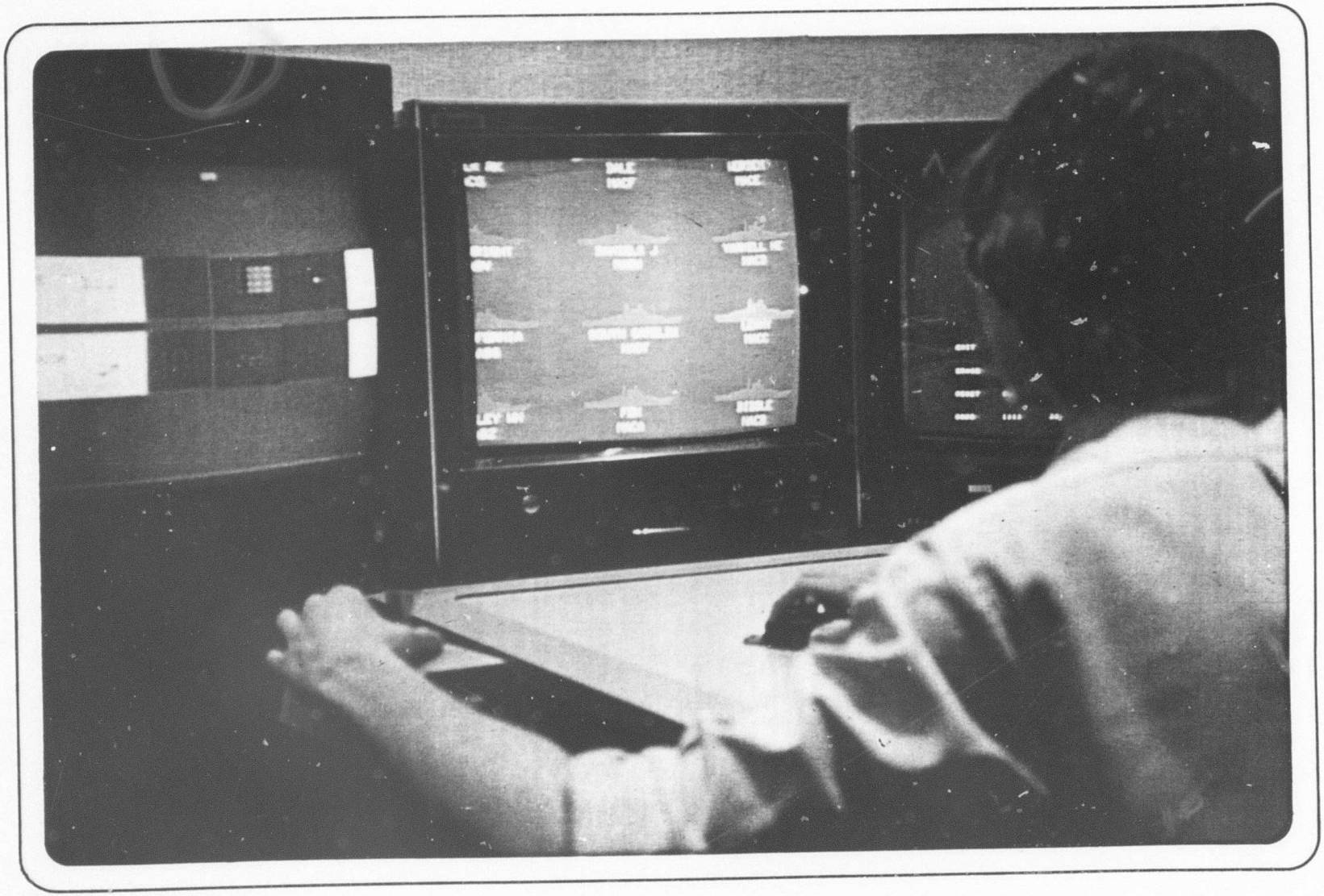


Figure 1. User Station

1.2 Control of Detail

During this quarter, the SDMS was augmented to permit the user to move in a direction perpendicular to the data surface. He can do this by twisting the joy stick, an action which changes the magnification at which the data is displayed, resulting in the zooming effect shown in Figures 2 and 3.

As the user continues to zoom the display, the system adds more detail to the picture, as shown in Figure 4. The user can continue to move across the I-Space, examining it at the new level of detail.

1.3 Image Planes

The addition of detail is implemented by storing the graphical representation of the I-Space at several levels of detail. These storage structures, known as image planes or i-planes are bit arrays which specify the color at each point on the I-Space at a given magnification. The program which allows the user to move through the I-Space adds more detail to the picture when the display is zoomed by substituting a more detailed i-plane for a less

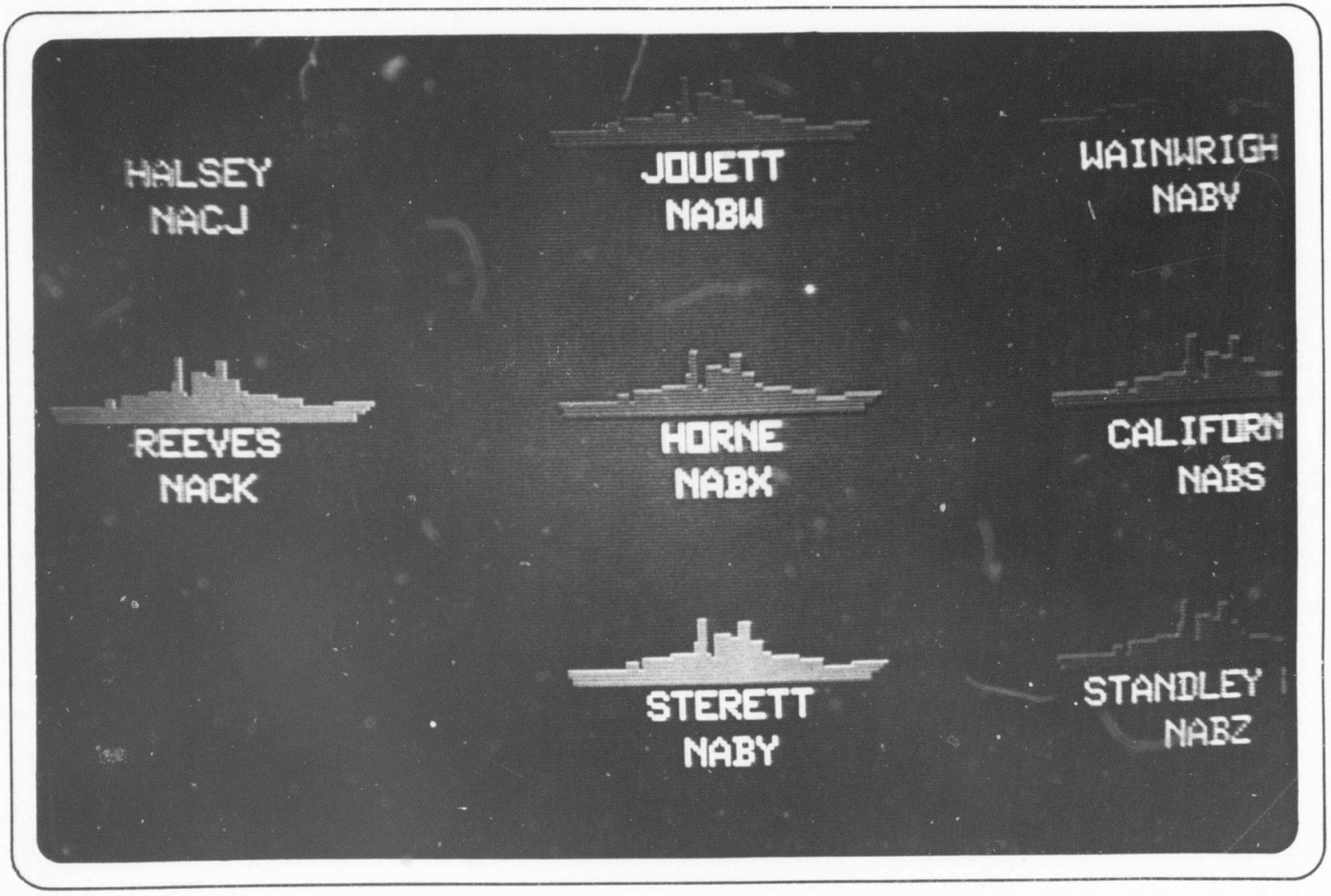


Figure 2 Main Display

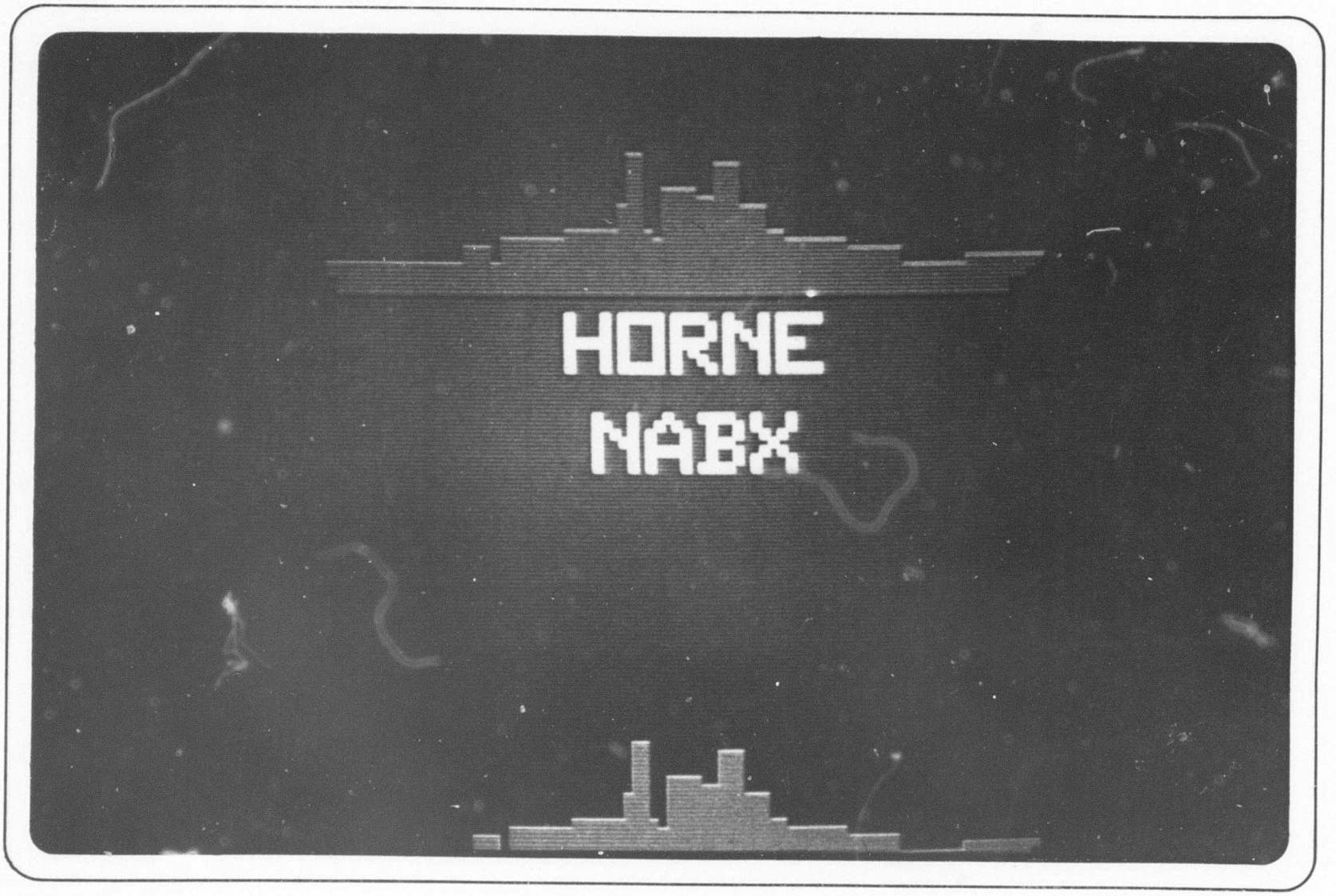


Figure 3 Main Display After Zooming

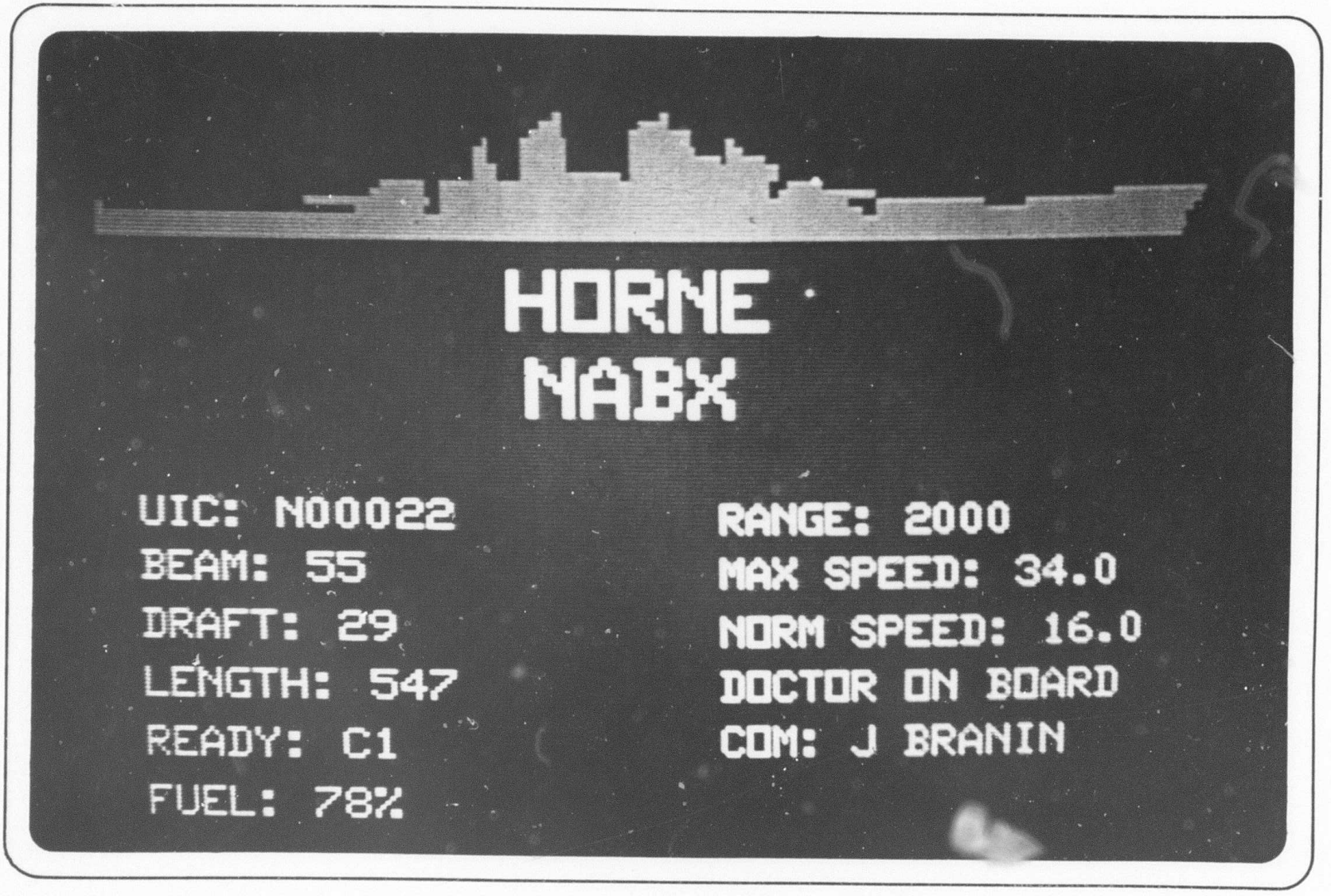


Figure 4 More Detailed View of Data Surface

detailed one.

1.4 Ports

The zooming motion is also used to control the movement between I-Spaces. Special areas of an I-Space, called ports can be designated as transition points between one I-Space and another. When the user zooms in on a port, the I-Space associated with that port becomes the current one. This mechanism can be used to partition the GDS into I-Spaces created for specific databases. For example, as user may create separate I-Spaces for ships, personnel files, and correspondence. The relationships among ports,

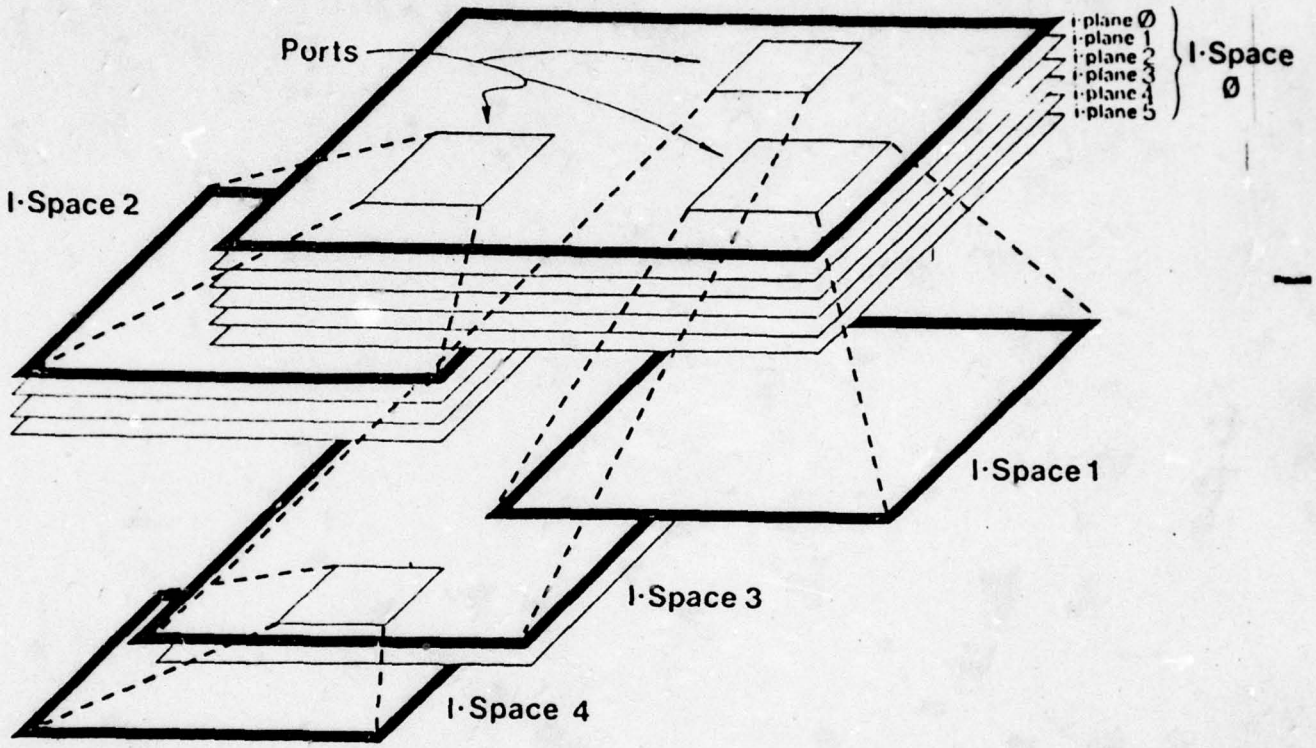
I-Spaces and i-planes is shown schematically in Figure 1.1

1.5 Graphical Displays of Symbolic Information

The icons in the I-Space of the preceding example were derived from data in a symbolic database management system (DBMS). The manner in which the appearance of each icon reflects its associated symbolic data can be controlled by

Nesting of I-Spaces Through Ports

Figure 1.1



the database administrator through a mechanism called the icon class description language (ICDL). At the beginning of this quarter, it was possible to define an icon's position and to display text along with it as a function of the symbolic data. During this quarter, it became possible to similarly define the shape, size, and color of an icon.

1.6 Integrity Maintenance

An important aspect of the graphical display of symbolic data is the facility for ensuring that the graphical display continues to reflect the symbolic data even after the database has been updated. This mechanism has been designed and will be implemented over the coming two quarters.

1.7 Overview

Chapter 2 of this report describes the additions to the motion programs to allow for motion between i-planes and between I-Spaces.

Chapter 3 describes the icon manager, the module which allocates space in a graphical data space and maintains the databases necessary to move through it.

Chapter 4 presents the design of the integrity maintentance module.

Appendix A contains the descriptions of the currently implemented statements of the icon class description language.

Finally, Appendix B provides details of the motion programs introduced in Section 2.

2. MOTION THROUGH THE GRAPHICAL DATA SPACE

There are three types of motion which the user can employ for controlling his view of the Graphical Data Space (GDS): scrolling, the goto, and zooming.

Scrolling is the means by which the user traverses an I-Space, moving his viewing position in a plane parallel to that I-Space. Scrolling was discussed in detail in the previous quarterly report and will not be covered here except to the extent that it is performed differently in order to allow the other two forms of motion.

While scrolling allows the user to move continuously over the data surface, there are times when it is advantageous to go directly from one point to another, without viewing the space in between. The goto provides the mechanism for such motion. It is discussed in detail in Section 2.3.

Finally, zooming allows the user to control the magnification of the view of the I-Space displayed on his CRT. Zooming is discussed in Section 2.2 below.

2.1 Motion Between I-Planes and between I-Spaces

A goto can be performed either explicitly or implicitly. An explicit goto is performed by the user typing in the name of the destination I-Space and coordinates, or by pointing to a location on the world view map.

Implicit goto's are invoked by zooming in on a port, in which case the *icon manager*, described in Section 3, returns the destination information to the motion program which then performs the goto implicitly (see Section 2.3.10).

2.2 Zooming

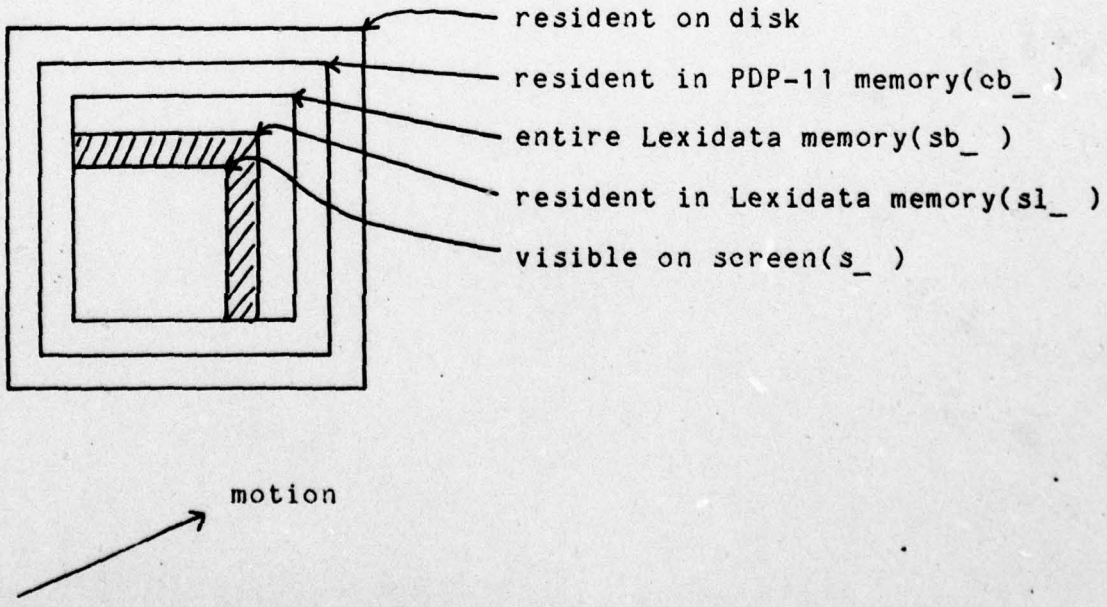
2.2.1 Introduction

The process of zooming the picture is separate from the work needed to replace one i-plane by a more detailed version, or "popped" view, although some of the mechanism of zooming is necessary for the smooth functioning of the i-plane transitions. This section is restricted to how the

proper image is maintained on the screen and how the supporting data is buffered in the display and in core.

Zooming only takes place when SDMS's motion system is in an acceptable state: the rectangles defining the screen (s_ variables), the loaded portion of the display's buffer (sl_), the full display buffer (sb_), and the core buffer (cb_) are properly nested and filled with valid pixel data. This situation is illustrated in Figure 2.1. In this design, display scale factors are restricted to values from 2 - 8, inclusive.

Nesting of Data Figure 2.1



Zooming is initiated from the joystick loop: when a rotation of the "Z" potentiometer on the joystick is noted, the current z speed is adjusted to reflect that change; and in every cycle in which z speed is non-zero, the routine zoom_image() is called. When the user reduces z speed to zero or reverses the direction of zooming, the routine zoom_clear() is called to ensure that all levels of the motion system's description of SDMS are consistent with the current display scale -- any pre-staging that may have been invoked in preparation for an expected zoom is canceled.

Zoom_image has its own set of bounds and state variables corresponding to those in the interprocess shared data area (common) which are manipulated to effect changes in scale; the new values are calculated and stored in these private variables until it is assured that the zoom can take place, and then copied from them into their corresponding public equivalents in common. These variables are named new_xxx, where xxx indicates the particular item, such as slx for the screen's left edge, sblen for the number of pixels across the screen buffer, or rows, for the number of tiles down the new tile map.

2.2.2 Overview of Zoom_image()

Zoom_image is called from do_run, the main loop of the navigator. It first calculates and validates the desired new scale:

if zspeed > 0, new_scale = current.dscale + 1
else if zspeed < 0, new_scale = current.dscale - 1

if new_scale < 2, goto previous iplane (currently a no-op);
else if new_scale > 8, goto next iplane (again no-op);
else the rest of this:

Call zdo_screen() to calculate the new values for the actual display, and set them if possible. This may fail, leaving the display unchanged, if in zooming out, the current sl_bounds are not big enough to hold the expanded screen. Any other failure is an error.

If outbound and zdo_screen succeeded, decrement new scale by 1; this allows preparation of the next buffer so the next zoom can take place when scheduled.

Call set_dimension() to calculate the dimensions of all the rectangles, as well selecting the next tile map; it sets the 4 remaining new variables (lx & ty for sb and cb) to the values of their current counterparts.

If inbound, call zdo_sbuf() to calculate the shrunken bounds for the new screen buffer. Then call zdo_cbuf() to shrink the core buffer around it. Otherwise (outbound), call zdo_cbuf first, to keep rectangles nested. In this case, zdo_cbuf may fail, because it cannot obtain sufficient buffer space for the expanded core buffer. If this happens, zoom_image fails (returns without modifiying any of the buffer variables). If it succeeds, zdo_sbuf is then called to expand the screen buffer into the newly opened space.

In either direction, zdo_slbuf adjusts the bounds of the loaded portion of the screen buffer to be consistent with the new sb bounds; if there is unloaded space, it queues feed requests to load it, and then move to move the sl_bounds out to meet the sb_bounds.

All the new variables are copied over into their correspondents in common.

As long as there are feed requests outstanding, do_feed() is called to process them.

Zoom_image returns to do_run.

The subroutines are discussed in more detail in Appendix B.

2.3 Goto

2.3.1 Overview

In order to go to to a new location in SDMS, we must

1. establish the destination precisely;
2. validate the destination and locate its graphical information;
3. assign buffer space in core and the display memory for it;
4. transfer appropriate data from disk to core and core to display;
5. reset the display and SDMS' internal status to reflect the new scene; and
6. release old buffers and flush the old state descriptors.

2.3.2 Establishing a Destination

When SDMS first starts up, it begins with a wired-in goto to the middle of the first i-plane of the 0th I-Space. Thereafter, gotos are performed at the request of the

user, who is prompted for a description of the destination. Zooms are targeted at the current position in an adjacent i-plane, as long as such exists. When no adjacent i-plane exists, a request to the icon manager results in a pair of destination descriptors being filled in Common, one for above and one below. These are the means by which a destination is obtained.

The information required for a destination consists of:

a. the I-Space -- given by a 16-bit I-Space id
b. the coordinates within the I-Space -- universal coordinates of the top left corner, as floats
c. relative i-plane in the I-Space -- an int index in the range 0 - 15
d. display scale to arrive at -- an int value from 2 to 8.

2.3.3 Validating a Destination & Locating its Pixels

Validating a destination involves locating its pixels; if the process cannot be completed successfully, the destination is not valid. The first step is to locate the I-Space descriptor, which contains access control information, the actual identifiers for the constituent i-planes,

and various other information. Descriptors are fixed-length records in a file, indexed by their identifiers so they may be read directly. If a descriptor exists for the desired I-Space and the access controls contained in it allow access by the current user, the i-plane id is retrieved for the desired level. If this exists, it is used as an index into a similarly-structured files of i-plane descriptors, and the corresponding record is read. This contains information on the disk location and encoding of the i-plane's pixels, as well as its dimensions and display characteristics. Pixels for an i-plane are arranged contiguously on the disk in "tiles" as described in [HEROT et al.] and their address can be computed from their coordinates and the information in the i-plane descriptor.

2.3.4 Allocating buffer space

Once the existence of the destination is guaranteed, the various areas corresponding to the screen itself, the surrounding frame buffer, and its surrounding core buffer, must be defined and allocated. The screen's dimensions are calculated according to the indicated display scale, and this rectangle is then positioned extending from the

desired upper left corner. It may be adjusted slightly to ensure that the screen lies entirely within the i-plane. Around the screen's rectangle is positioned a screen buffer, again with a size computed from the scale, and position depending on the i-plane dimensions and destination coordinates. Farther out, a larger rectangle is defined for the core buffer, similarly constrained as to size and position. Each of these rectangles is defined in terms of pixels; that is, it extends over the defined area of the i-plane, without reference to where in core or the display buffer those pixels are stored.

An additional complication is added here by the fact that a moving from one I-Space to another. implies replacing the navigational aid's display, as well as the scene on the main display. This may cause difficulties for a zoom leading to a pop-through: If the direction of motion is down (inward), the navigational aid's buffers will be smaller than the new display's, and the allocation process described next may proceed without consideration of the needs of the nav aid. But for backing out of an I-Space, the nav aid will generally require more space than is available, and the zoom-pop sequence must be suspended until it may be executed as a simple goto.

Buffer space in the display is calculated by lines outside those currently committed to the current view. Even

though large portions of each line used in the current view may not be committed to it, they cannot be allocated to to the new view, because sequential addresses will pass into the beginning of the following line before reaching its free area, thereby destroying information in the current scene. The wraparound nature of addressing in frame buffer does allow lines to be assigned on both sides vertically of a scene which resides in the middle of the buffer.

Buffer space in core is allocated in tiles of 8K bytes each; these are located in the large core buffer, and managed by a routine which arbitrates requests from the navigator, picture construction, nav aid, and the interface routines to the display.

2.3.5 Loading Buffers

Assuming buffers are defined and allocated successfully, the process of putting up the new picture begins. This actually may involve putting up two pictures, since a change in I-Space will involve displaying a new navigational aid, as well as a main display. The navaid uses buffers temporarily, to move data to the screen, but once

displayed, does not update it, and so may release its buffers to the main display process.

Data is transferred from disk to core, and from core to the display, using exactly the same mechanisms as for scrolling. That is, a request is sent to the diskio process indicating a desired tile and the core buffer in which to store it, in addition to the status flag in which to mark its completion. The fact that this buffer may be committed to a view in preparation rather than the current one is completely transparent to these routines. From core to the display, areas corresponding to rectangles in the i-plane are broken into subareas restricted to a single tile; a request to feed each of these is formatted and dropped in the queue. At the end of such a sequence of feed requests, another request moves the appropriate bound(s) to indicate that that portion of the display buffer is now loaded.

In performing any of the versions of a goto, the feeds are broken into two parts: the first just covers the area which will be displayed on the screen, after which the display may be switched to actually show that scene. The remainder of the display buffer (the margins which support smooth scrolling) may be fed after the switch is made, although no scrolling is possible until they have been completed. This distinction allows the scene to be

changed once 70% of the display buffer has been filled with the new scene.

2.3.6 Switching the Display

All the state information referring to the status of the display, location of buffers, and their state, is kept in duplicate -- one copy for the current scene, and one for any that may be in preparation. The latter area is used for collecting all information in preparation for a goto (of whichever variety). When the scene has been appropriately prepared, it can be displayed by swapping the state variables, and calling the routine which resets the display processor to whatever parameters are contained in the current state variables.

2.3.7 Cleanup after a Goto

After the swap of state variables mentioned in the preceding section, all information relating to the old state is available in the "next" structure. The core buffers associated with this state's tile map are returned to the

buffer manager and their tile blocks cleared, and the bounds of the various rectangles and display parameters are all set to -1. The I-Space and i-plane descriptors are not cleared, on the chance that they may be useful in the next goto. (This is especially likely for the I-Space descriptor in a zoom through multiple i-planes of an I-Space.)

2.3.8 Special Processing for a Pop-Through

Pops differ from simple goto's in three respects:

1. They are initiated by the user's motion, rather than an explicit request; and therefore
2. their destination is calculated or supplied by a request to the icon manager rather than provided by the user; and
3. they must maintain the current view while preparing the new one.

This last requirement may involve scrolling or zooming the current scene, with the associated data transfers among disk, core, and display buffers.

All processing for the current scene is triggered and synchronized by a regular loop in which the joysticks are

read every 30 ms.; if the current scene is not to be abandoned during preparation of the new, then that preparation must be fit in and around this fixed cycle. This period is far too short to accomplish the complete setup, so it must be broken into small pieces which can be performed independently in a single cycle on the joysticks, and a mechanism provided for properly sequencing the performance of those parts.

This mechanism consists of a small executive function which maintains an indicator of which step is to be taken next, and incorporates the schedule that processing those steps must follow. The analogy is straightforward to a simple cpu whose program counter indicates the next setup step to be processed, and whose basic clock tick corresponds to reading the joysticks once. Departures from that analogy are discussed below.

Recognizing the need for a pop-through and acquiring its destination are the other two areas of difference from a simple goto. Performing a zoom (described in Section 2.2) triggers the setup for a pop; in the inward direction, any zoom higher than scale 2 is taken as evidence of the user's intent to pop into another scene; in the reverse motion, the major work is involved in zooming out to lower scales of a single scene, so the pop processing is not initiated until the user reaches scale 3.

When it appears that the user may need to pop into another i-plane, the destination must be found. If a further i-plane exists in the current I-Space in the indicated direction, that will be the destination, with the new scene in the same universal coordinates, and at the opposite extreme of scale (current defaults are 2 and 6). If no such i-plane exists, a request is sent to the icon manager to determine whether the user is located over a port. After searching its database, the icon manager fills in two structures in shared core, one indicating the destination if the user moves down, and one for up, from the current position; these structures also indicate how far the ports extends, so the icon manager need not be bothered with further queries while one answer remains valid. If no port exists, the icon manager fills in the destination structure with the value -1, and the navigator simply stops when it reaches the maximum zoom in that direction.

2.3.9 Failures and Cancellations

Any form of goto may fail because the requested I-Space or i-plane does not exist, or because insufficient buffer space is available to set up the new scene. Non-existent

I-Space and i-planes are recognized when the attempt is made to retrieve their descriptors. This takes place before any resources such as buffers are committed to the new scene, so a failure simply results in cancellation of the goto without any backtracking.

In a simple goto, lack of buffers leads to restoration of the current view (possibly including its nav aid) and display of an explanatory message on the user's console. In the normal case, the amount of buffer space required to restore a view cannot be more than what was acquired by releasing that view's buffers in the first place, the restoration of view is generally possible. (The exception occurs in a situation in which a nav aid must be redisplayed for a higher scale view of an i-plane; the amount of core buffer space needed temporarily for the nav aid may exceed that used by the main display, and the difference may be temporarily allocated to some other process such as picture construction. This situation should not arise in the current system because no other process requires enough buffers to prevent restoration of a nav aid, and, independently, the current nav aid is not currently destroyed until sufficient buffer space has been obtained for the goto to succeed.)

In the zoom/pop case, buffer space is generally not available at all times; some amount is tied up supporting the

current display. In the case of a zoom in, this amount is decreasing steadily, so that after the current display reaches scale 4, there is generally enough to support the scene in preparation as well as the curent one. In the opposite direction, the amount of space required for the new scene is small enough to be compatible a scale 2 image on the current display; the only difficulty in assigning the buffer area earlier is ensuring that the two do not overlap. In either case, however, extraordinary situations may prevent buffer allocation at the usual time. If this happens, the zoom/pop must be delayed until a later stage of the zoom (higher scale), or, ultimately, until it can be treated as a simple goto, succeeding or failing finally just as a simple goto would.

If user motion in the z-direction initiates setup processing for a pop, but the user then reverses direction or begins to scroll horizontally, the preparation for a pop is cancelled, on the theory that the destination is no longer what was calculated initially. Resumption of the original Z motion re-initiates preparation for a pop, but of course, the user may be starting from a higher scale than usual, so some delay may be evident before the pop is executed.

Cancellation before buffers are allocated is free is performed as follows. The new I-Space and i-plane

information is retained, since it has not destroyed any information needed for the current scene, and may yet prove useful. If buffers have been allocated to the next view, they must be released; if they were taken at the expense of the current view, they must be reloaded with curent information. Cancellation that occurs after the screen has been set to the new view is ineffective; it is simply taken as a scroll on the new scene, or a goto in the reverse direction.

2.3.10 ZOOM_CYCLE

Zoom_cycle is called on each joystick loop in which z_speed is non-zero. At each invocation, it determines if there is work that can be done in preparing for a pop-through into a new location, and if so does a portion of that work calculated to fit in the 30 ms window dictated by the joystick cycle. Various conditions may cause the expected pop to fail or be cancelled. Zoom_cycle is responsible for cleaning up the remains and leaving things in an acceptable state if this happens. If the destination is a process rather than a location in information space, zoom_cycle is responsible for invoking the process, waiting for its termination, and then restoring the state of the display.

Zoom_cycle is essentially a switch on the value of the zoom_agenda word in common. This is initialized to the value ZREADY, and manipulated by the various functions of zoom_cycle; it corresponds to the program counter of zoom_cycle. A static destination structure (corresponding to those in common which refer to ports) is used to maintain information about an adjacent i-plane destination across invocations; a static pointer is used to indicate which destination descriptor is relevant. Other information built up about the destination is maintained in the

"next" state_variables structure in common.

Discussion of the specific states of zoom_agenda, and the processing invoked for each is described in Appendix B.2.

3. THE ICON MANAGER

3.1 Function

The function of the Icon Manager is two-fold. One function is that of a database management system for information pertaining to icons, I-Spaces, and regions. The second function is the management of space in the Graphic Data Space (GDS).

The Icon Manager contains information about every icon in the GDS such as: its origin, extents, color, whether it has been named, whether it is a port, and pointers to entries in parallel databases. I-Spaces are treated as special icons, and have additional information stored. Regions within I-Spaces can be allocated and are considered as sub-I-Spaces. Details concerning design and implementation issues are discussed in the following sections.

The Icon Manager allocates space for new icons within an I-Space or region. When the requested location of a new icon would overlap the region or I-Space, the nearest non-overlapping location is returned. It also determines

where existing icons are located and whether an arbitrary point in any I-Space is contained within an icon.

The Icon Manager runs as a separate process in SDMS and is called by other processes such as the navigator, the hierarchy map process, the association processor, integrity maintenance, picture construction, and the graphical editor.

3.2 Design

3.2.1 Hierarchy of Data

The Icon Manager maintains a strict hierarchy among the objects that it manages. That hierarchy is the following:

1. I-Space
2. Region
3. Icon
4. Subicon

An I-Space is a data surface which may consist of several planes (i-planes) of graphical information. A region is a user-defined rectangular portion of an I-Space which may

be used to target the results of an association. An icon is a pictorial representation of some datum or collection of data in the symbolic database. It occupies a rectangular area in an I-Space on one or more of the i-planes which comprise the I-Space. Subicons are rectangular areas within an icon. They are to an icon what an icon is to a region and/or I-Space.

An I-Space can contain either regions, icons, or both. A region can contain only icons. An icon may contain subicons. Note that there need not be regions within an I-Space, nor are subicons required within an icon. Such functions are supported but their use is optional.

When a new I-Space is created, an entry is made in a list of I-Spaces and in the master list of icons. A unique I-Space id is assigned to the new I-Space. Thereafter, when regions or icons are created in that I-Space, they must be targeted within the bounds of the I-Space. An attempt to place the icon or region outside the bounds of the I-Space causes an error condition.

Regions are useful for the logical subdivision of an I-Space. The user might, for example, divide an I-Space into three regions and target the results of three different associations to each of the three regions. The icon manager guarantees that no icons targeted to a region will

be moved outside that region to avoid collision with existing icons.

When an icon is created, it is targeted to a location in an I-Space. That location is examined to determine whether an existing icon overlaps that area. If so, the icon is moved within the region, if there is one, and/or I-Space until an unoccupied location is found. If there is no room within the region or I-Space, an error condition is returned.

Icons are the fundamental unit of graphical information in the GDS. They may be subdivided using subicons to convey more detail. A ship icon, for example, might be subdivided into subicons for the hull, flag, and superstructure. These subicons can then be modified by the appropriate rules in the Icon Class Description Language (ICDL).

3.2.2 Database Format

The Icon Manager's database consists of a number of Unix files which are managed solely by the Icon Manager. Each of these files contains records pertaining to a class of objects maintained by the Icon Manager. Files exist for the following kinds of objects:

- I-Spaces
- Regions
- Icons
- Ports
- Names

The format of these files is discussed below.

3.2.2.1 The I-Space File

The I-Space file contains one record for every I-Space which currently exists. In addition, information is stored relating the largest valid I-Space identifier assigned, a count of deleted I-Space records, and a pointer to a linked list of deleted I-Space records. Each record within the file contains the I-Space id for that record and the icon-id which corresponds to that I-Space. The icon-id serves as a pointer to the corresponding record in the icon file, where detailed information about the I-Space may be found. The I-Space id serves as an index to the appropriate I-Space record.

3.2.2.2 *The Region File*

The region file is identical in format to the I-Space file. Each record contains a unique region-id and a

pointer to the corresponding record in the icon file. In addition, a count of records, a count of deleted records, and a pointer to a linked list of deleted records is maintained. Like the I-Space file, the region-id is used as an index to the appropriate region record.

3.2.2.3 The Icon File

The icon file contains the essential information which the Icon Manager needs to perform its duties. The I-Space and region files simply act as maps, mapping I-Space or region id's into icon-id's. The real information is contained in the icon records. Icon records contain the following information:

Origin of icon in universal coordinates
Extents of icon in universal coordinates
Parent of icon
Child of icon
Sibling of icon
Type of icon
Pointer to corresponding port record
Pointer to corresponding search record
I-Space which contains icon
Color of icon
Background color of icon

Pointer to corresponding name record

The origin of the icon is the coordinates in x, y, and z of the upper left corner of the icon. The extents of the icon are the dimensions of the icon in x, y, and z.

The parent of the icon is the icon-id of the smallest object enclosing the icon. I-Spaces have no parents, so this value is irrelevant for an icon whose type is I-Space. A region's parent is the I-Space which contains it. A simple icon (one whose type is icon) has either a region or an I-Space as its parent. A subicon has the icon which contains it as its parent.

The child of an icon is the icon-id of an icon (of any type) which is contained by that icon. The child is actually a pointer to the beginning of a linked list of all icons which have the same parent. An I-Space's child may be either a region or a simple icon. An empty I-Space, of course, has no children. A region's child can only be an icon, and an icon's child can only be a subicon. A subicon's child can only be another subicon.

The sibling of an icon is the icon-id of the next icon in the linked list of the parent's children. Hence, the parent icon record has the icon-id of one of its children. That child is in turn linked to another child of its

parent by its sibling entry. By following the list of siblings starting at the child of an icon, one can find all icons which have the same parent.

The type of an icon is one of: I-Space, region, simple icon, subicon, or port. Ports and simple icons are identical, except that a port contains a pointer to an entry in the port file.

Only an icon of type port has a valid pointer to a port record. In all other icon types this field is initialized to a default value.

Icons of all types except I-Space and region have pointers to records in the corresponding search file. The function of the search file will be described in a following section.

All icons contain a field which is the the I-Space id of the I-Space in which the icon is found.

The color of an icon is stored in order to distinguish the area of prime interest from any background that might be contained within the bounds of the icon. This value is only valid for icons whose type is simple icon. The background color is the color of the area where the icon is placed. This is saved so that when the icon is erased, the proper color will be used to overwrite the icon.

An icon of any type may be named. If the icon has been named, the pointer to the corresponding name record will be filled. Otherwise, it contains a default value.

In addition to the information contained in each icon record, the icon file also contains information such as: the last icon-id assigned; the count of deleted icon records; and a pointer to a linked list of deleted icon records. The icon-id is used as an index to reference the appropriate icon record.

3.2.2.4 The Port File

The port file contains additional information about icons whose type is port. A port is a gateway to a point in the same or another I-Space, a specific icon in the GDS, or a Unix process. The port record has the format:

Type of port
Length of record in bytes
Deleted flag
Information specific to each type of port

The port type can be one of: point, icon, or unix. The length is the total length of the record. The deleted flag signals whether the information in the record is valid.

Point ports contain the following additional information: target I-Space id; universal coordinates (x, y, and z) of a point in that I-Space; and the scale to which the display should be set upon popping through.

Icon ports contain the following additional information: the icon-id of the icon which is to be displayed; and the scale to which the display should be set upon popping through.

Unix ports contain the following additional information: the name of the Unix process to be run upon popping through; and two arguments to that process if required.

In addition to the information contained within the port records, the port file contains the last port-id assigned and pointers to deleted records of each port type.

3.2.2.5 The Name File

The name file contains a list of names which have been assigned to icons. Name records have the format:

Name

Icon-id

Tree links

The name of an icon may be up to 15 ASCII characters in length. No case mapping is performed, so upper- and lower-case names are distinctive. The icon-id is a pointer to the icon record to which the name belongs. The tree links are pointers to other name records. The name file is maintained as a binary tree to allow efficient searching for and insertion of name records.

3.2.3 Spatial Database Format

In addition to the information database which contains information about the meaning and function of an icon, the Icon Manager also maintains a spatial database. The spatial database allows the user to map between spatial coordinates and entries in the icon file.

The spatial database consists of a number of files, one for each I-Space in the GDS. Within each file are records for each icon in that I-Space which is not not of type I-Space or region. These files are referred to as search files. The records within each file are sorted on the basis of the coordinates of the icon's origin. This allows an efficient search for an icon given the coordinates of a point within the icon.

The spatial database is used when determining whether an existing icon contains a given point in the I-Space. This is of value when determining what icon the user is pointing to on a display and in positioning new icons (the findspace problem).

The format of records in a search file is as follows:

Origin of icon

Extents of icon

Icon-id

Links

The origin and extents of the icon are the same as are stored in icon records. The icon-id is a pointer to the corresponding record in the icon file. The links point to the next search records in the sorted file.

The search files contain records which are sorted on first the y-coordinate, and then the x-coordinate of the icon's origin. This has the effect of dividing the I-Space into rows of icons. Only those rows which have y-origins less than or equal to the search point's y-value need be searched. The search of a particular row need only continue until an x-origin greater than the search point's x-value is found.

3.2.4 Manipulation of the Database

Standard operations on the database include retrieving a record, adding a new record, deleting a record, and updating an existing record. Retrieval is a straightforward process, assuming the appropriate index in the appropriate file is known. Addition, deletion, and updates, however, are more complex in that several files must be modified and kept consistent with one another. These operations are the subject of this section.

It is more revealing to discuss these operations in terms of the various types of icons that are being manipulated, since different icon types will cause different files to be referenced.

3.2.4.1 I-Spaces

Addition of an I-Space is relatively simple. First, a new I-Space id must be assigned. If there are deleted I-Space records, the I-Space id at the head of the list is assigned and the deleted list and counter are fixed. Otherwise, the I-Space count is incremented and the new value assigned. Next a new icon-id must be assigned. The same procedure is followed as for the I-Space id. Deleted icon-id's are re-used whenever possible. The I-Space id

and icon-id are inserted into an I-Space record and written to the I-Space file. Now the icon record is filled with the origin and extents of the I-Space, the icon type is set to I-Space, and the remaining values are defaulted. The icon record is then written to the icon file.

Deletion of an I-Space is a straightforward process. The relevant I-Space record is marked as deleted and chained into the list of deleted records. The updated record is written to the I-Space file. The icon record pointed to by the I-Space record just deleted is retrieved. If the I-Space has no children, the record is marked as deleted and chained into the list of deleted icons. If the I-Space had children, then it is necessary to follow the chain of siblings, deleting them (see Section 3.2.4.3). If any icon in the chain has children, then they, too, must be deleted. If the I-Space was named, the name must be deleted from the name file (see Section 3.2.4.5). Once this has been completed, the I-Space's icon record can be marked deleted and chained into the list.

The most common update of an I-Space is to change its size. This is accomplished by modifying the appropriate icon record. One must make sure, however, that no children of the I-Space will be outside the new bounds. This entails a check of all children of the I-Space in the manner described above. Icons which would be outside must

be deleted from the appropriate files.

3.2.4.2 Regions

Since regions and I-Spaces are logically the same, the same procedure is followed when adding, deleting, and updating region records as for I-Space records.

One additional constraint when adding or updating a region is that the new region must fit within the bounds of its containing I-Space.

3.2.4.3 Icons

Adding a simple icon requires more than the physical insertion of a new record in a file. The proposed icon must be subjected to the findspace process (described in Section 3.2.5). This guarantees the icon will not overlap any existing icon. Once that is done, a new search-id must be assigned in the search file for the appropriate I-Space. Deleted search records are re-used whenever possible. A new icon-id must also be assigned. Deleted icon records are used whenever possible. This done, the search record is filled with the origin and extents of the icon and the icon-id. It is then inserted into the proper spot in the sorted lists of the search file. The icon record

can then be filled in.

The icon record is filled with the origin and extents of the icon. The parent is filled with the icon-id of the icon record of the containing I-Space. If the parent has a child, the new child becomes the new icon, and the new icon's sibling becomes the old child of the parent. Otherwise, these values are defaulted. The child value is defaulted. The type of the icon is loaded. The I-Space id and search-id for the icon are loaded. All remaining values are defaulted. The icon record is then written to the icon file.

To delete a simple icon, the appropriate icon record is retrieved. If the icon was named, the name record must be deleted (see Section 3.2.4.5) The search record for the icon must be deleted. This involves updating the sorted linked lists in the appropriate search file. If the icon has a child, it must also be deleted. If the icon's child had siblings, they and any of their children must also be deleted. Once all offspring have been deleted, the icon itself can be marked deleted, and chained into the list of deleted icons. Deletion can become a recursive process when there are children and grandchildren.

Updating an icon usually means changing the size of the icon or moving it. In the case of a change in size, if

the icon becomes smaller, then all that need be done is to change the extents in the icon record and in the appropriate search record. If the icon grows in size, the search record for the icon must be deleted, and the icon resubmitted to the findspace process. The results of the findspace process are then loaded into a new search record and inserted into the search file. The new extents are loaded in the icon record. If the icon had to be moved by findspace, then the process described below must be executed.

When an icon is moved, the icon record is loaded with the new origin. The icon's search record must be deleted and a new one inserted with the icon's new origin. If the icon had children, then the origin stored in their icon records must be updated with the relative change in position. The search records for all the children must be deleted and new ones inserted with the updated origins.

3.2.4.4 Ports

Adding a port consists of making an existing simple icon into a port. This is accomplished by assigning a new port-id for the appropriate type of port. If there is a deleted port record of the proper variety available, then it is re-used. The port record is filled with the

appropriate values and then written to the port file. The icon type is changed to port and the port pointer is saved. The icon record is then re-written.

Deleting a port is accomplished by stripping the icon of its status as a port. The port record is linked into the list of deleted port records. The icon type is changed to icon and the port pointer is defaulted. Both port and icon records are written back to their respective files.

Updating a port means changing the values stored in the port record. The port record is retrieved, the new values loaded, and the updated port record is written back to the port file.

3.2.4.5 Names

Adding a name first requires that the icon to be named doesn't already have a name. If it does, an error return is taken. If not already named, the name file must be searched to ensure that the new name is not already used. If the new name is not in the search tree, a new name record is created with the name. It is then loaded with the icon-id of the newly named icon. The record is then written into the name file. The icon record is updated with a pointer to the corresponding name record.

Deleting a name requires searching the name file for the desired name. If it is found, the icon record to which the name belongs is retrieved and the name pointer cleared. The name record being deleted is then patched into a list of deleted name records. Then the binary search tree must be fixed. This is done according to the algorithm suggested by Knuth, Vol. 3 [KNUTH].

Updating a name requires that the appropriate name record be deleted, and a new one created. If the new name is already in use, an error return is taken.

3.2.5 Findspace

Findspace refers to two separate tasks. One is the detection of overlaps in the GDS. This is used for two purposes: determining if a point is contained within an icon; and determining whether a proposed icon would overlap any existing icon. The second task is moving an icon which overlapped an existing icon to a portion of the region and/or I-Space where it doesn't.

Using the mechanism of search files which contain sorted lists of all icons within an I-Space, the detection of overlap is simple and efficient.

To find which icons contain a given point in an I-Space, it is necessary to compare the origin and bounds of each icon in the I-Space with the coordinates of the point. However, by using sorted lists of icons based on their origins, it is possible to reduce the search to a small subset of the icons in the I-Space. It is necessary to examine only those icons which have both y-origin and x-origin less than or equal to the corresponding coordinates of the point. The search files are structured as sorted lists of sorted lists. The initial list is sorted on y-origin and the other lists are sorted on x. Therefore, a search need only examine those lists which have y-origins less than or equal to the y-coordinate of the point. Each list need only be examined until an x-origin is found greater than the x-coordinate of the point. The search terminates when an icon is found containing the point or an icon with a y-origin too large is found.

An almost identical scheme is used to find whether a proposed icon would overlap existing icons. The y-list is examined until an icon is found with a y-origin greater than the bottom y-value of the proposed icon. The x-lists are searched until a x-origin greater than the right x-value of the proposed icon. A check is performed on all icons in the search file which meet these qualifications to make sure that no point of the proposed icon is within

the existing icon.

If a proposed icon would overlap an existing icon, it must be moved to avoid the collision. The Icon Manager moves colliding icons in a spiral pattern around the target location of the proposed icon. This allows the icons generated by an associate to be clustered around the target location. When an overlap is detected, the Icon Manager moves the icon to the next position in the spiral, and examines that location for overlap. The process repeats until an unfilled slot in the spiral is found. That becomes the location for the new icon. If the spiral would take an icon outside the bounds of the target region or I-Space, that slot is ignored and the pattern followed until the spiral re-enters the region or I-Space. If there is no room in the I-Space or region, an error condition is returned.

4. GRAPHICAL VIEWS OF SYMBOLIC DATA

A key feature of the CCA SDMS prototype has been the automatic generation of graphical views of symbolic data. This symbolic data is maintained in the INGRES relational database management system [HELD STONEBRAKER WONG]. The graphical views are collections of pictures called icons which represent tuples in INGRES. The appearance and placement of an icon is dependent on the contents of the tuple it represents.

The database command language of SDMS, called SQUEL for Spatial QUEry Language, is a superset of QUEL, the INGRES command language. SQUEL contains two commands which invoke the automatic generation of icons. The ASSOCIATE command creates a dynamic picture of a relation in which updates to a tuple subsequent to its association will result in graphical updates to the icon which represents it. The mechanisms which monitor changes in the symbolic database and bring about the required changes in the graphical database are referred to as Graphical Data Space (GDS) integrity maintenance.

The other command which causes automatic icon generation, the DISPLAY command, produces a static view of the database. GDS integrity maintenance does not apply to icons

produced by the DISPLAY command.

Both the ASSOCIATE and the DISPLAY commands must specify an Icon Class Description (ICD). An ICD is a set of rules which specify the appearance of an icon as a function of the contents of a tuple. A tuple and the icon representing it are referred to as an entity tuple and its entity icon.

4.1 ASSOCIATIONS

The SQUEL associate command is invoked with two arguments, the name of a relation and the name of an icon class description. This command directs SDMS to create an entity icon for each tuple in the specified relation according to the rules of the icon class description. The tuple that a particular icon represents is referred to as the icon's entity tuple.

Subsequent symbolic data changes in an entity tuple will cause appropriate graphical data changes in its entity icon through the mechanisms of graphical data space (GDS) integrity maintenance. Association processing involves three SDMS processes:

1. the SQUEL interpreter, which recognizes SQUEL com-

mands;

2. the association processor, which is the background controller of association processing; and
3. icon creation - picture construction, which actually manipulates the GDS.

When a valid associate command is recognized in the SQUEL interpreter, the association processor is sent the necessary information to begin the association. This information consists of the contents of the tuples to be associated and the name of the icon class description. When association processing is complete, the association processor provokes maintenance of the association map relation in the SQUEL interpreter. The association map relation is the SDMS system relation which provides a map between entity tuples and their associated entity icons.

4.1.1 The association map

The maintenance of the correspondence between entity tuples and entity icons is implemented by several table-driven routines collectively known as graphical data space (GDS) integrity maintenance. The driving table is the association map relation. Maintained by the SQUEL interpreter as an SDMS system relation, the association map

relates entity icons to entity tuples and the icon class descriptions used to create them. The actual domains of the map are: relation, tuple-id, icon-id and icon class description. Tuple-ids are INGRES assigned identifiers that are unique only to the tuple's relation, hence, the association map's relation and tuple-id domains are both necessary to specify a unique entity tuple. Icon-ids are SDMS assigned identifiers unique to the entire GDS. By mapping relation names to invalid icon-ids, the association map also serves to flag any associated relations for which the icon-ids are presently unavailable. This situation occurs whenever an association is in progress as icon-ids are inserted into the association map only after association processing is complete.

All association map maintenance is performed by the SQUEL interpreter. Most entries to the map are made following association processing. The association processor done flag is consulted upon each activation of the SQUEL interpreter. A set flag indicates that new entity icon-ids are available and should be entered into the map with their entity tuples before further SQUEL processing.

4.1.2 Implementation overview

Initiation of association processing involves the copying the associated relation from INGRES storage structures to the standard UNIX file rel.SDMS. The association processor is then invoked with two parameters, the name of the icon class description and the name of a file containing a description of the relation. This file provides the association processor with the information necessary to "read" tuples from rel.SDMS. In the event that a relation description does not exist at the time of the association, the SQUEL interpreter creates one before association processor invocation.

The purpose of the association processor is to allow entity icon construction to proceed concurrently with other SDMS operations. Its specific function is to repeatedly invoke icon creation - picture construction (once for each entity tuple in rel.SDMS) and to save the returned icon-ids in the UNIX file icons.SDMS.

The icon creation - picture construction process manipulates the GDS at icon granularity. Its functionality includes icon creation and erasure. Arguments to this process are the contents of a tuple (passed via the UNIX file stfile) and the name of an icon class description. An icon class description is a set of rules describing the

appearance of an icon as a function of the contents of its entity tuple.

When association processing is complete, the association processor will set its AP-done flag. When consulted by the SQUEL interpreter, this flag will be reset, and the icon-ids in icons.SDMS will be read into the asociation map relation. At this point, the association map entry which signals the unavailability of icon-ids for this relation will be removed.

4.2 DISPLAY COMMAND AND ERASE COMMAND

The SQUEL display command differs from the associate command in two important ways: an optional qualification allows the selection of specific tuples, and the graphical view of the relation is static rather than dynamic. The static nature of display icons means that they are never automatically erased from the GDS. Because of this, displays are named by the user for subsequent use in the SQUEL erase command.

In its implementation, the display command utilizes the same icon generating mechanisms as the associate command. GDS integrity maintenance is bypassed for these icons by

saving the display icon-ids in the display map rather than in the association map. The display map relates display names to icon-ids.

The SQUEL erase command takes a display name as its argument. When recognized by the SQUEL interpreter, an Integrity Maintenance Daemon (IMD) erase request is made for each icon mapped to the specified display name in the display map. An IMD erase request causes GDS integrity maintenance to erase the specified icon (see Section 4.3 below).

4.3 GDS INTEGRITY MAINTENANCE

Graphical Data Space (GDS) integrity maintenance deals with the problems of maintaining a faithful representation of the symbolic database in the GDS. Specifically, the appearance of each entity icon must change as its entity tuple changes. Addition or deletion of tuples to an associated relation must affect addition or deletion of entity icons in the GDS.

GDS integrity maintenance involves three main subtasks:

- identification of symbolic database operations which may affect GDS integrity,

- background control of integrity maintenance operations, and
- manipulation of the graphical data space.

Identification of relevant symbolic database operations is done by the SQUEL interpreter, through which all symbolic database interactions pass. Identifications of Ingres append, replace, delete, and destroy commands which apply to any associated relation causes (queued if necessary) requests to be made to the Integrity Maintenance Daemon (IMD). IMD requests are of three types: create icon, recreate icon, and erase icon. The form of an IMD request is a message packet in the IMD request queue.

Each IMD request packet contains all the data necessary for the integrity maintenance daemon to direct the completion of one IMD request. Queuing of these requests allows background operation of most integrity maintenace processing. To the user, GDS integrity maintenance appears to proceed concurrently with all other SDMS operations, including subsequent symbolic database interactions.

Manipulation of the GDS is through icon creation - picture construction, a process which is capable of constructing and erasing icons.

4.3.1 Implementation overview

The association map relation contains a list of all associated relations. SQUEL add, replace, delete, and destroy commands specifying any of these relations affects GDS integrity maintenance, and causes the SQUEL interpreter to issue IMD requests.

The addition of a tuple to an associated relation requires the addition of an entry to the association map relation. This entry maps the new tuple's tuple-id to its icon-id. New tuples' icon-ids are provided by the icon manager. A create icon request is sent to the IMD. Create icon request packets contain: an icon-id, the name of the relation, and the contents of the new tuple. These data are utilized by the IMD in formulating the icon creation - picture construction invoking parameters which will cause the construction of a new entity icon.

Replacing a tuple of an associated relation produces an IMD recreate request similar in format to that of the create request. In response to a recreate request, however, two invocations of icon creation - picture construction are made by the integrity maintenance daemon. The first invocation erases the old entity icon, the second invocation recreates it as specified by the tuple's new contents. Deleting a tuple of an associated relation or

destroying an associated relation results in IMD erase requests from the SQUEL interpreter. Erase request packets contain only the icon's icon-id. In response to an erase request, the IMD directs integrity maintenance - picture construction to erase the entity icon.

Concurrent association processesing, and GDS integrity maintenance initiated by a replace, delete, or destroy operation on the same relation, is not possible. Recreate and erase IMD request packets include the entity icon's icon-id which is found in the association map relation. As icon-ids are inserted into the map only after a successful association, recreate and erase IMD requests cannot be made until the association is complete. In these situations, processing of the replace, delete, or destroy command will be suspended until the association is complete and the icon-id data is available.

APPENDIX A

NAME

template icon

SYNOPSIS

TEMPLATE ICON number

DESCRIPTION

Selects the template icon from the template image plane.

NAME

attribute region statement - defines an attribute region.

SYNOPSIS

ATTRIBUTE REGION attribute_name FROM (x1,y1) TO (x2,y2)

DESCRIPTION

Defines an attribute region and specifies its position within the icon. The region will display the value of the specified attribute.

DEFINITIONS

<attribute> ::=
ATTRIBUTE REGION<attribute_name>
FROM (<arith_expr>,<arith_expr>)
TO (<arith_expr>,<arith_expr>) ;

<attribute_name> ::= <identifier> ;

EXAMPLE

```
/* attribute region for persons phone */

   attribute region r.phone from (0,0) to (50,20)
```

SEE ALSO

general(icdl), plane(icdl), example(icdl)

DIAGNOSTICS

BUGS

NAME

image plane statement - statement to describe one plane of an icon.

SYNOPSIS

IMAGE PLANE plane# plane_stmts END

DESCRIPTION

The image plane statement fully describes one plane of an icon for an icon class description. The plane# states which plane in the I-Space is intended. The plane_stmts further modify this picture. They include text, coloring, drawing pictures, etc.

The icon used for a plane should be one plane deep. If the icon covers more than one plane, only the top-most plane is used.

DEFINITIONS

<plane> ::= IMAGE PLANE <plane#> <icon_id> <plane_stmts> END ;

<plane#> ::= <arith_expr> ;

<plane_stmts> ::= <plane_stmt> |
<plane_stmt> <plane_stmts> ;

<plane_stmt> ::= | <attribute region> |

EXAMPLE

see example(icdl)

SEE ALSO

general(icdl), icon(icdl), example(icdl)

NAME

position - defines target position for icon.

SYNOPSIS

POSITION (x,y)

DESCRIPTION

Defines the target position for each icon. The target position is the position in the I-Space where SDMS will attempt to place the created icon. This position is in the user's coordinates for the I-space. If it cannot place it at the target position, it attempts to find a position close by. If the icon cannot fit anywhere, an error occurs.

DEFINITIONS

<position> ::= POSITION (<arith_expr>,<arith_expr>) ;

EXAMPLE

```
/* set icon origin to be I-Space origin
   (taken from example(icdl))                */

   position (0,0)
```

SEE ALSO

general(icdl), icon class(icdl), example(icdl)

DIAGNOSTICS

BUGS

NAME

color statement - performs color filling on a region of the icon.

SYNOPSIS

COLOR OF REGION region_number IS color

DESCRIPTION

The specified portion of the template will be filled with the specified color. The regions of a template are defined when the template is created by flooding each of them with a different color. The region_number is the same as the index of the color used to flood that region. In this was arbitrary shapes can be colored.

DEFINITIONS

<color> ::= COLOR OF REGION<arith_expr>,
<color_expr> ;

EXAMPLE

COLOR OF REGION 2 IS green

SEE ALSO

DIAGNOSTICS

BUGS

NAME

maximum size - defines the maximum size of the icon.

SYNOPSIS

MAXIMUM SIZE IS (width,height)

DESCRIPTION

Defines the maximum size of the icons generated by the icon class. If the generated icon is greater than this size, scaling takes place automatically to reduce it to the maximum size.

DEFINITIONS

<maximum> ::= MAXIMUM (<arith_expr>,<arith_expr>) ;

EXAMPLE

```
maximum (+100,+120)
```

SEE ALSO

NAME

scale statement - scales the icon within the limits of the icon class definitions.

SYNOPSIS

SCALE IS s

DESCRIPTION

Scales the icon on the plane being described. If s>1 then the icon increases in size. If s<0, an error occurs. Otherwise, the icon decreases in size. In any case, the scaling is limited to be between the minimum and maximum sizes of the icon.

DEFINITIONS

<scale> ::= SCALE BY <arith_expr> ;

EXAMPLE

```
/* double the icon size */

   scale by 2
```

SEE ALSO

APPENDIX B

B.1 DO_IMAGE Subroutines

B.1.1 DO_SCREEN()

Calculate the new screen length and height as

new_slen = (SCRN_LEN / new_scale) - 32

new_slen = SCRN_HT / new_scale;

reduce either as necessary to make it fit the iplane.

Calculate an xdelta and ydelta as half the difference between new and old dimensions, positive if the dimension shrank (inbound case).

Calculate the new upper left corner of the screen as the old corner plus the deltas in both x and y. If this point lies outside the current sl_ boundary, shift it down and/or right until it is safe.

Calculate the new lower right screen corner as the upper left plus the new length or height, minus 1. If this lies

outside the current sl_corner, shift up/left to make it fit. If that moves the upper left corner out, the current buffer will not hold the new image; return -1.

Otherwise, copy screen bounds, dimensions, and scale in to common; call set_current() to send these new parameters to the display; return 0;

B.1.2 SET_DIMENSIONS()

Set new sb dimensions from the appropriate array in navdat.c, indexed by new_scale.

Select the new_map and dimensions (cols and rows) according to the value of new_scale; restrict cols and rows to be less than or equal to the tile dimensions of the current iplane.

Compute new cblen and new cbht as cols and rows times the tile dimension, but restrict them to be less than or equal the corresponding iplane pixel dimension.

Copy from current into new_sblx, _sbty, _cblx, and _cbty. All new_ variables are now initialized. Return.

B.1.3 DO_CBUF()

Compute x and y deltas as half the difference between new and current cb_ dimensions, positive for the inbound case.

Compute the new upper left bounds as the current value plus the appropriate delta, truncated to a tile boundary; reset negative values to 0; if the new value is greater than the new upper-left corner of sbuf (equal to the old one if we haven't calculated a new yet), shift up/left until it is valid.

Compute the new lower right bounds as the new upper left plus the new dimensions minus one. If this lies outside the iplane by more than a tile dimension, shift all bounds left/up until valid. If it lies outside the new sb lower right corner, shift down/right until valid. If this leaves the upper left corner inside the sb bounds, abort -- cb dimensions have not been chosen large enough for this scale.

If inbound, call copy_dn with the new_map, the number of rows to skip at the top of the map, and the number of columns to skip at the left of the map. (These are computed as new_bound/TILE_dimension - current.bound /TILE_dimension.) Copy_dn does not have an error exit.

Return.

B.1.4 DO_SBUF()

Compute x and y deltas as half the difference between new_sb dimensions and those in current; positive values for the inbound case.

Compute the new upper left bounds as the current bounds plus the appropriate delta, truncated to the minimum feed width (16 across, 4 down). Shift down/right while outside the new_cb upper left bounds (equal the current values when inbound, because zdo_cbuf hasn't been called yet). Shift left/up while inside the new (= current) s_ upper left corner. No need to test against cb again, because tile boundaries are multiples of the sb increments, and s is nested with cb.

Compute the new_cb lower right bounds as upper left plus the new dimension minus 1. If this leaves new sb lower right outside cb, set it to cb. If sb is inside s lower right, shift right/down until valid. If this invalidates the upper left corner, abort -- sb dimensions have been chosen wrong for this scale.

Return.

B.1.5 ZDO_SLBUF()

In turn for the top, bottom, left, and right sides:

If new_sl lies outside new_sb, set it to new_sb; else If strictly inside sb, feed the rectangle defined by the three outer sb bounds and 1 outside the inner sl bound. Reset the sl bound to the corresponding sb bound.

B.1.6 COPY_DN(new_map, tskip, lskip)

A rectangle of tiles contained in the current tile map is to be copied into the new map, filling it. The remainder of the old map is defined by tskip rows on the top, (old_rows - new_rows - tskip) rows on the bottom, and in each of the intervening, shared rows, lskip tiles on the left, and (old_cols - new_cols - lskip) tiles on the right. The tiles in this remainder have their buffers returned to the buffer pool and their t_blocks cleared. Copying tiles is accomplished by swapping the pointers in corresponding locations in the two maps.

Compute rskip and bskip.

```
For tskip rows
  for old_cols tiles
      add tile's t_buffad to list to be returned
      fill t_block with -1's

For new_rows rows
  for lskip tiles
      flush tile as above
  for new_cols tiles
      swap pointers from old and new maps
  for rskip tiles
      flush tile as above

For bskip rows
  for old_cols tiles
      flush tile as above

call free_buf(old_size - new_size,
  -1,                          /* don't need to swap these into an
  list of buffer addresses,    /* active map, as for paint;
  same list)                   /* therefore also don't care what the
                               /* tile-id list says
```

B.1.7 COPY_UP(new_map, tskip, lskip)

Like copy_dn, but this time the source is smaller than the destination. So instead of flushing tiles, we allocate new ones: buffers must be requested from the buffer manager; tile_ids computed, and i-plane & status filled in each new block, and diskio queued for that block.

Upper left tile id of dest is computed as upper left tile id of source, minus lskip, minus (tskip * ip_xtiles). Number of buffers to request is computed as the difference of the two maps' sizes; if the buffer manager cannot supply them, copy_up fails (returns -1). Rskip and bskip are computed as for copy_dn.

```
For tskip rows
    for newcols tiles
        init_block:   t_id      = base id + column
                      t_i-plane  = current i-plane
                      t_status  = TBUSY
                      t_bufad   = next from list
                                  buffer manager returned
                      get_tile(t_block)
    add ip_xtiles to base id

For old_rows rows
    for lskip tiles
        init_block as above
    for old_cols tiles
        swap pointers from old and new maps
    for rskip tiles
        init_block

For bskip rows
    for new_cols tiles
        init_block
```

B.2 Zoom_agenda states

ZREADY: No setup processing has been accomplished. The first step is to find if a destination is to be used, and if so, which one.

call find_dest(); dispatch on its return code:

FAIL: can't go that way from here. leave agenda untouched; return.

SUCCESS: destination pointer refers to a valid destination. set agenda to ZGETIS; return.

PROCESS: destination is a process rather than a location. set agenda to ZSPAWN; return.

ZGETIS: call get_is with the I-Space id in the regnant destination struct. Returns are:

FAIL: no such I-Space. set agenda to ZREADY; complain; return.

SLOW SUCCESS: a new I-Space descriptor was read from disk. set agenda to ZGETIP; return.

FAST SUCCESS: I-Space descriptor found without i/o.
set agenda to ZGETIP; retry switch.

ZGETIP: call get_ip with the level number in the destination. Returns are:

FAIL: no such i-plane. set agenda to ZREADY; complain; return.

SLOW SUCCESS: a new i-plane descriptor was read from disk. set agenda to ZNEWSCREEN; return.

FAST SUCCESS: i-plane descriptor found without i/o. set agenda to ZNEWSCREEN; retry switch.

ZNEWSCREEN: if destination pointer refers to static struct, call new_corner to figure the universal coordinates of the upper left corner of the destination.

if (inbound and scale < 4) or (outbound and scale > 2) buffer space isn't ready yet; return immediately.

call news_screen with left x, top y, and desired scale; returns are:

FAIL: screen buffer space not available. set agenda to ZWAIT; return.

SUCCESS: the "next" struct has its various display parameters set correctly, including an available buffer in the display. set agenda to ZNEWLCB; retry switch.

ZNEWLCB: call new_lcb. returns:

FAIL: core buffer space not available; have to wait for more: set agenda to ZWAIT; return.

SUCCESS: the "next" struct has its core buffer parameters correctly set, including reservation of sufficient pages of buffer space. set agenda to ZNAVAID; retry switch.

ZNAVAID: if new and old I-Space id's equal, set agenda to ZLOAD; retry switch.
else call set_navaid for next I-Space; returns:

FAIL: set agenda ZWAIT; retry switch.

SUCCESS: set agenda to ZLOAD; return

ZWAIT: if current scale not yet at extreme, return; call goto_image with the destination values. returns:

FAIL: call restor_current; set agenda to ZREADY

SUCCESS: set agenda to ZREADY; return (you're in the destination).

ZLOAD: call load_lcb with the tile id stored in next.tile0. set agenda to ZFEED1; return;

ZFEED1: call feed_image for the main display, next-structure, and bounds of the screen in the next structure. call FEED_BND to set the bounds of the loaded area the same as the bounds of the screen. (Neither of these actually accomplishes that effect; it simply queues the appropriate request.) Save the end of this sequence of requests. set agenda to ZFEED2; return

ZFEED2: if requests from ZFEED1 remain in the queue, do 2 of them. if the feed queue has been emptied as far as the end of the sequence generated by ZFEED1, set agenda to ZCOMMIT. return.

ZCOMMIT: if current scale not at the extreme or zoom clock > 0, return. call swap_state; call set_current; call flush_state on the next structure; set agenda to ZFEED3; return

ZFEED3: call feed_image to fill in the margins above, below, left, and right of the screen in the screen buffer; call FEED_BND to move the loaded bounds appropriately. set agenda to ZREADY; return.

References

[HELD STONEBRAKER WONG]
Held, G.D.; Stonebraker, M.R.; Wong, E., "INGRES - A relational data base system", AFIPS Proceedings, Volume 44.

[HEROT et al.]
Herot, C.F.; Kramlich, D.; Carling, R.; Friedell, M.; and Farrell, J., "Quarterly Research and Development Technical Report, Spatial Data Management System", Computer Corporation of America, 575 Technology Square, Cambridge, Mass., 02139 (December 1978).

[KNUTH]
Knuth, Donald E., The Art of Computer Programming, Volumn 3, "Sorting and Searching", Addison-Wesley, Menlo Park, California, pg. 429 (1975).